Pairwise Coverage-based Testing with Selected Elements in a Query for Database Applications

Koji Tsumura, Hironori Washizaki, Yoshiaki Fukazawa
dept. Computer Science
Waseda University
Tokyo, Japan
kojisuke_awesome@asagi.waseda.jp

Keishi Oshima, Ryota Mibe
Hitachi, Ltd., Research & Development Group, Center for Technology Innovation - Systems Engineering
Kanagawa, Japan
keishi.oshima.rj@hitachi.com

***Abstract*—Because program behaviors of database applications depend on the data used, code coverages do not effectively test database applications. Additionally, test coverages for database applications that focus on predicates in Structured Query Language (SQL) queries are not useful if the necessary predicates are omitted. In this paper, we present two new database applications using Plain Pairwise Coverage (PPC) and Selected Pairwise Coverage (SPC) for SQL queries called Plain Pairwise Coverage Testing (PPCT) and Selected Pairwise Coverage Testing (SPCT), respectively. These coverages are based on pairwise testing coverage, which employs selected elements in the SQL SELECT query as parameters. We also implement a coverage calculation tool and conduct case studies on two open source software systems. PPCT and SPCT can detect many bugs, which are not detected by existing test methods based on predicates in the query. Furthermore, the case study suggests that SPCT can detect bugs more efficiently than PPCT and the costs of SPCT can be further reduced by ignoring records filtered out by the conditions of the query.**

***Index Terms*—combinatorial testing; database testing; coverage; test data**

I. Introduction

Testing database applications using components based on database management systems is difficult because the execution path and behaviors depend on the database state. Although some code coverages (e.g., statement coverage, branch coverage, etc. [1][2]) and testing methods automatically generate tests, prioritize test cases, and localize bugs based on code coverage [3][4][5][6], they cannot detect bugs due to the records in the databases because data variations in the records are not considered.

Most applications maintain data using a Relational Database Management System (RDBMS) and a Structured Query Language (SQL) query to manage the contained data. Some coverage and test methods consider data variations with SQL executed in the applications [7][8][9], but they focus on the condition predicates in the query such as in the JOIN or WHERE clauses. These methods can detect bugs caused by conditional mistakes in the queries [e.g., misplaced logical operators (AND, OR) or mistakes of conditional inversion]. However, they cannot detect bugs due to the lack of conditions such as an omission of a necessary condition in the SQL query or in the IF statement in the program code.

To detect bugs due to a lack of conditions, we propose Plain Pairwise Coverage Testing (PPCT) and Selected Pairwise Coverage Testing (SPCT), which are new testing methods for database applications using Plain Pairwise Coverage (PPC) and Selected Pairwise Coverage (SPC), respectively, for SQL queries. Pairwise testing [10] is a Combinatorial Test Design technique in which two parameters and their test values are taken as inputs to test all combinations of two parameters' values. In PPC, all COLUMNs with parameters in the TABLEs in the select query are used as the parameters, and the test values are set to verify if all possible combinations of the two parameters' values appear in the query results. In SPC, COLUMNs in the SELECT query are used as the parameters, and all their test values are set to verify if all possible combinations of the two parameters' values appear in the query results. Furthermore, SPC focuses on the WHERE clauses to remove combinations that cannot be obtained by the query.

This study aims to address the following Research Questions:

RQ1 Can PPCT and SPCT detect bugs that existing testing method cannot?

RQ2 How many records are needed to reach 100% PPC and SPC coverage?

To evaluate the applicability of PPCT and SPCT, we use a case study where PPCT and SPCT are applied to two open source software systems [11] [12]. Each open source software system has about 400KLOC and interacts with about 100 tables and 1000 columns. Because they have bug management systems, we employ their issues as target bugs. In our case study, PPCT and SPCT detects 61 of the 89 bugs that existing method cannot. In addition, the case study results suggest that ignoring records not obtained by the query can further reduce costs.

This paper makes the followings contributions:

- PPC and SPC, which are new record coverages that apply pairwise testing coverage using selected elements in a query, are proposed.
- A tool to calculate PPC and SPC is implemented.
- PPCT and SPCT, which are new test methods for database applications using PPC and SPC, respectively, are proposed.

- PPCT and SPCT are applied to two actual products.
- PPCT and SPCT can detect 68.54(%) of bugs that existing testing method cannot.
- SPCT can detect bugs more efficiently than PPCT by ignoring redundant combinations.

The remainder of the paper is organized as follows. Section II provides background information and a motivating example. Section III describes PPCT, SPCT, and our tool to calculate PPC and SPC. Sections IV and V present our case study and results, respectively. Section VI discusses related works. Finally, Section VII contains our conclusion and future works.

II. Background

Table I shows the schema, while Figure 1 shows the code and query of our motivating example. Every record in the Table `item`, which indicates whether the record is alive or not, has the column `voided`. The method `sumPrice(String category)` calculates the total price of the 'unvoided' items. This code finds items by category (Line 4) and sums up their prices (Line 6-10). Because the code sums only 'unvoided' item's prices, 'voided' items in the query must be filtered out. However, 'voided' items are not filtered because there is neither WHERE clause in the query nor an IF statement in the code. Thus, records must be prepared according to specific criteria to test this code.

A. Testing with Code Coverage

Consider a test that uses code coverage (e.g., statement coverage). In the motivating example, we only need `sumPrice("food")` for the `item` record whose `category` = `'food'` to achieve 100% statement coverage. However, this method cannot detect the abovementioned bug with a record whose `category` = `'food' AND voided` = `false` even though the statement coverage is 100%.

B. Testing with Predicate Coverage of SQL

Some studies test database applications with SQL [7][8][9] by focusing on the predicates of SQL such as conditions in the WHERE clause and the integrity constraints of the database schema. These criteria can detect conditional mistakes in the query because they verify whether record variations can be covered in terms of the conditions. However, they cannot detect bug in the motivating example because it is due to the lack of a condition.

Consider a code test using the "Full Predicate Coverage"(FPC) proposed by Tuya et al. [8]. This coverage focuses on the conditions of the JOIN clause and the WHERE clause in the query to evaluate the MC/DC coverage of the query. For the motivating example, FPC focuses on the condition `WHERE category = 'food'`. To achieve 100% coverage, FPC requires records whose `category` = `'food'` and `category` ≠ `'food'`. However, even with 100% coverage, it cannot detect bugs with records `voided` = `false`.

TABLE I
Database Schema of the Motivating Example

Table `item`		
Attribute	Type	Constraint
id	Int	Primary Key
category	String	Not null
voided	Bool	∈ {true, false}
price	Int	Not null
created_at	Date	Not null
updated_at	Date	Not null

```
1  /* Calculate a total price of 'unvoided' items */
2  public sumPrice(String category) {
3    // Get 'item' records from database with query
4    List<Item> items = getItems(category);
5    int price = 0;
6    for (Item i : items) {
7      // Sum up prices of 'unvoided' items
8      // Bug!!! We need filtered out 'voided' items, for
   example, adding IF statement 'if (!i.isVoided())'
9      price += i.getPrice();
10   }
11   return price;
12 }

/* Query executed by "getItems(category)" */
SELECT
id, category, voided, price
FROM item
WHERE category = 'food';
```

Fig. 1. Code and query of the motivating example

C. Pairwise Testing

Combinatorial testing can detect bugs due to parameter interactions in the software being tested with a covering array test suite generated by an algorithm. Pairwise testing is a combinatorial testing technique in which two parameters and all their test values are used as inputs to test all possible combinations. If only one or two conditions exist, more than 70% of bugs can be detected [13]. Because pairwise-generated test suites are smaller than exhaustive ones and effectively find defects, many tools exist to generate pairwise test suites [14].

To test database applications with a pairwise testing technique, the parameters are (tables or columns), candidate values, and presence of constraints all must be consider.

III. Pairwise Coverage-based Testing

PPCT and SPCT are new test methods for database applications that focus on the variety of database records used in testing. To reduce the number of tested combinations, we use a pairwise testing technique.

A. Overview

Figure 2 overviews PPCT and SPCT. The target of PPCT and SPCT is the database used in the application being tested. Three inputs are required: the target database for the coverage measurement, a test case to extract the SQL SELECT query, and the test criteria. The last one is retrieved from the

requirement documents, database schema definitions, and the tendency of the records in the actual database. PPCT uses PPC (plain pairwise coverage), while SPCT uses SPC (selected pairwise coverage) for a query. PPC and SPC generate the coverage criteria using a pairwise testing technique which uses columns from the query as parameters and equivalence classes of columns as candidate values. The query results are used to verify which coverage criteria are satisfied. Both PPCT and SPCT involve six steps. In our coverage measurement tool, steps 3 - 5 are automatically implemented.

1) Execute a test case:
Testers retrieve the desired SQL SELECT query to measure the coverage from DBMS query logs or application logs.

2) Set the candidate equivalence classes of the columns in the database:
Testers can set the following equivalent classes based on the type of the column:

Bool
: Testers can set 'true' and 'false' as candidate values.

String
: Testers can set what to measure in a regular expression manner.

Boundary Value
: Testers can set the range of the values they want to measure with comparison operators (i.e., $=$, $\neq$, $>$, $\geq$, $<$, or $\leq$)

3) Execute a query:
Our tool executes the SQL SELECT query obtained by step 1 to the database in the initial input and collects the query results.

4) Generate the coverage criteria:
Pairwise testing is used as an element technology to generate the coverage criteria. Our tool generates the coverage criteria using the equivalence classes of the columns created in step 2 and the SELECT query in step 3. Details of this step are described in the next subsection.

5) Measure the coverage:
Our tool measures the coverage using the query results in step 3 and the coverage criteria in step 4. Details of this step are described in the next subsection.

6) Update the database:
According to the coverage measurement results in step 5, records are inserted, updated, and deleted in the database.
Steps 3 - 6 are repeated to prepare an effective database set for testing.

B. Our Record Coverages

In this subsection, we propose our record coverages to test the database applications used in steps 4 and 5 in the PPCT/SPCT overview. Figure 3 an example of the flow for steps 4 and 5 using the motivating example in Section II.

1) Plain Pairwise Coverage (PPC):
In PPC, we focus on the all COLUMNs with TABLEs in the query.

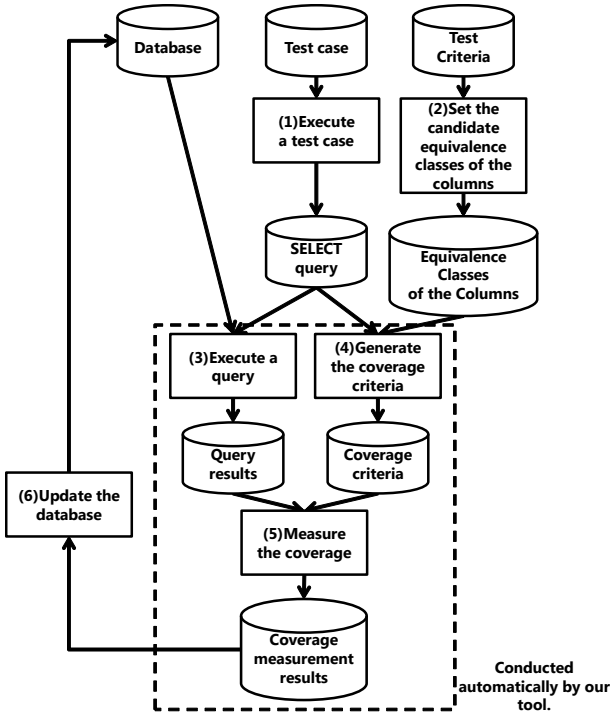

Fig. 2. PPCT/SPCT overview

q represents a SQL SELECT query obtained by step 1 in the PPCT/SPCT overview. If $c_i (i = 1, 2, \cdots)$ represents the parameter column used in pairwise testing, then C (the set of parameter columns) is defined as $C = \{c_1, c_2, \cdots\}$. For the example in Figure 3, the schema of the table `item` is shown in Table I. Because PPC focus on all COLUMNs in TABLES referred in the query q, $C = \{\texttt{id}, \texttt{category}, \texttt{voided}, \texttt{price}, \texttt{created\_at}, \texttt{updated\_at}\}$.

Let us refer to the equivalence classes of columns as the user definition obtained by step 2. If $d_j (i = 1, 2, \cdots)$ represents a column in the user definition, then D (the set of columns in the user definition) is defined as $D = \{d_1, d_2, \cdots\}$. In Figure 3, $D = \{\texttt{category}, \texttt{voided}, \texttt{created\_at}\}$. In the user definition, if $e_{d_j k} (k = 1, 2, \cdots)$ represents an equivalence class of column d_j, then E_{d_j} (the set of equivalence classes of column d_j) is defined as $E_{d_j} = \{e_{d_j 1}, e_{d_j 2}, \cdots\}$. In Figure 3, $E_{\texttt{voided}} = \{true, false\}$.

For each column $c_i \in C$, the set of possible values of c_i is defined as a set of equivalence classes E_{c_i}, which is given by the following formula:

$$E_{c_i} = \begin{cases} E_{d_j} & (c_i \in D \wedge c_i = d_j) \\ U_{c_i} & (otherwise) \end{cases} \quad (1)$$

U_{c_i} in formula (1) means an equivalence class that includes any possible value for column c_i. In Figure 3, the column `id` is not in D. Therefore, $E_{\texttt{id}} = U_{\texttt{id}}$. Because the column `id` is an integer type, $U_{\texttt{id}} = \{x | x \in \mathbf{Z}\}$. Between columns c_i and $c_j \in C$, the set of combinations of possible values to be

tested P_{ij} is defined by the following formula (where $i > j$ and $N = |C|$)

$$P_{ij} = U_{c_1} \times \cdots \times U_{c_{i-1}} \times E_{c_i} \times U_{c_{i+1}} \times \cdots \times U_{c_{j-1}} \times E_{c_j} \times U_{c_{j+1}} \times \cdots \times U_{c_N} \quad (2)$$

Then the set of all possible combinations with at least two parameters to be tested P is defined by the following formula (where $N = |C|$)

$$P = (\cup_{i=1}^{N-1} \cup_{j=i+1}^{N} P_{ij}) \bigcup (\cup_{k=1}^{N} E_{c_k}) \quad (3)$$

P for the flow example is shown as the coverage criteria in Figure 3.

If the set of the parameter pairs appearing in the results obtained in step 3 is referred to as $Appeared(P)$, then the coverage Cov_{PPC} is defined as

$$Cov_{PPC} = \frac{|Appeared(P)|}{|P|} \times 100(\%)$$

In the coverage measurement results in Figure 3, a check (cross) indicates that the pair does (does not) appear in the records. In this example, step 3 gives two records, but only three combinations, $\{(\texttt{voided}, \texttt{category}) = (false, `food'), (\texttt{voided}) = (false), (\texttt{category}) = (`food')\}$ appear. Hence, $Cov_{PPC} = 3/18 = 16.67(\%)$. However, in this example, the parameter combination $(\texttt{voided}, \texttt{category}) = (false, `clothes')$ cannot appear in the records because it is filtered out by the WHERE condition `category = 'food'`. Furthermore, parameter combinations concerned with the column `created_at` will not appear in the records because the column is not selected by the query. Therefore, PPC cannot reach 100 % and PPC is not effective in some cases.

2) Selected Pairwise Coverage (SPC): PPC focuses not only on the parameter combinations obtained by the query, but also the parameter combinations not obtained. In contrast, selected pairwise coverage (SPC) focuses on parameter combinations that can be obtained by the query to cover them efficiently. SPC regards the set of all COLUMNs selected in the query q as C. In Figure 3, $C = \{\texttt{id}, \texttt{category}, \texttt{voided}, \texttt{price}\}$. The definitions of E_{c_i}, P_{ij}, and P are the same as those of PPC.

To ignore parameter combinations not obtained by q, P must be filtered. Let W be the set of equivalence classes of columns satisfying the conditions in q. In Figure 3, $W = \{E_{category} = \{`food'\}\}$. $P_{satisfied}$, which is the parameter combinations that can be obtained by q, is defined as the set calculated by the algorithm in Figure 4. The dotted lines in Figure 3 show the set of combinations for $P_{satisfied}$. The set of pairs appearing in the results obtained by step 3 is referred to as $Appeared(P_{satisfied})$. Finally, the coverage Cov_{SPC} is defined as

$$Cov_{SPC} = \frac{|Appeared(P_{satisfied})|}{|P_{satisfied}|} \times 100(\%)$$

In Figure 3, $Cov_{SPC} = 3/5 = 60(\%)$.

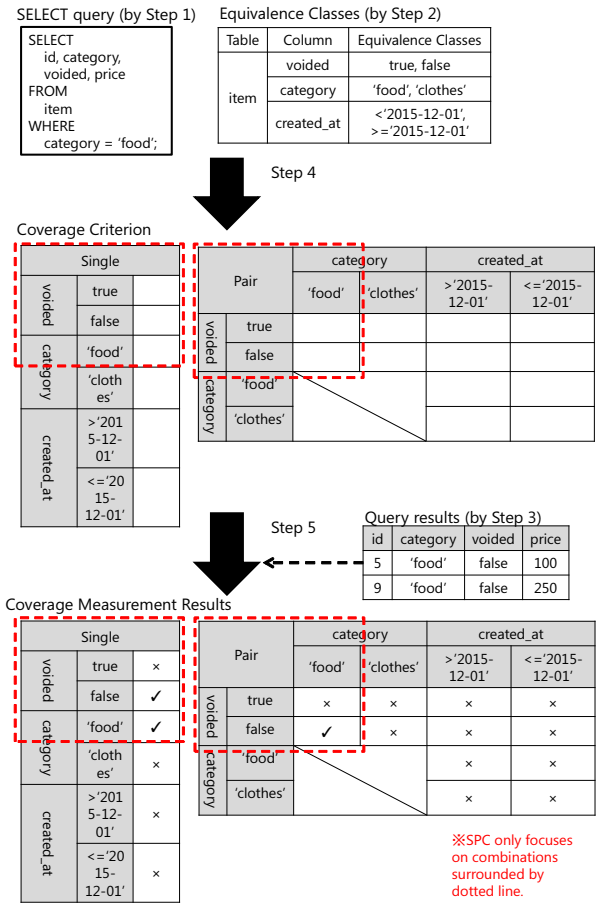

Fig. 3. Example of the flow in steps 4 and 5

```
P_satisfied = ∅
for (P_ij in P) {
  for (p_ij in P_ij) {
    if (p_ij ∩ W != ∅) {
      P_satisfied = P_satisfied ∪ p_ij
    }
  }
}
for (E_pk in P) {
  for (e_pk in E_pk) {
    if (e_pk ∩ W != ∅) {
      P_satisfied = P_satisfied ∪ e_pk
    }
  }
}
```

Fig. 4. Algorithm of $P_{satisfied}$

C. Testing with our record coverage

Next, we tested the motivating example mentioned in Section II with our record coverage.

Figure 3 shows the parameter combinations in PPC. Among them, a bug is found when calling `sumPrice("food")` with the record where $(\texttt{voided}, \texttt{category}) = (true, `food')$

The necessary parameter combinations in SPC are also shown in Figure 3. Similar to PPC, a bug is detected when calling `sumPrice("food")` with the record where $(\texttt{voided}, \texttt{category}) = (true, `food')$

IV. Case Study

We carried out a case study, targeting *OpenMRS* [11] and *Broadleaf Commerce* [12]. These targets were chosen because:

(1) They use SQL queries to manage the data in their databases.
(2) They have real bugs, which are individually managed using a tracking system.
(3) They have a sufficient number of tables and columns to evaluate RQ2.

A. Case Study Overview

OpenMRS is an open source medical record system platform for developing countries. We used the *openmrs-core* module, which is a core module of *OpenMRS* that has api and web application codes as the targets. *Broadleaf Commerce* is an open source e-commerce platform. We used the *Broadleaf-Commerce* module, which is a core module of *Broadleaf Commerce*. Both are written in Java®[1] and interact with approximately 100 tables in the database. We used *MySQL*®[2] (Ver 14.14 Distrib 5.5.27, for Win32) as the RDBMS. The case study used four versions of *OpenMRS* and one version of *Broadleaf Commerce* for the control system *Git* [15]. Figure 5 overviews of the case study, which can be divided into five steps:

1) Systematically filter bugs:
Issues are filtered by condition. *OpenMRS* manages the bugs and features to be implemented in the future by *JIRA* [16], which is an issue management system. Initially, issues are filtered by the condition "Project=TRUNK AND IssueType=Bug AND Status=Closed". The condition "Project=TRUNK" removes bugs not related with *openmrs-core*. If an issue is closed, the cause of the bug can be determined by inspecting the discussion and activities of the issue. Next issues are extracted by the committed files with at least one file related to database access. Service classes to execute SQL to obtain records, controller classes to process the records obtained through service classes, and their test classes are assumed to be related to database access. Because *Broadleaf Commerce* manages the bugs and features to be implemented in the future by *Git* issues page, issues are also filtered by the condition "label:type-bug is=closed". Next issues are extracted by committed files as well as *OpenMRS*. All of these files can interact with the database directly or indirectly.

[1]Java is a registered trademark of Oracle and/or its affiliates.
[2]MySQL is a registered trademark of Oracle and/or its affiliates.

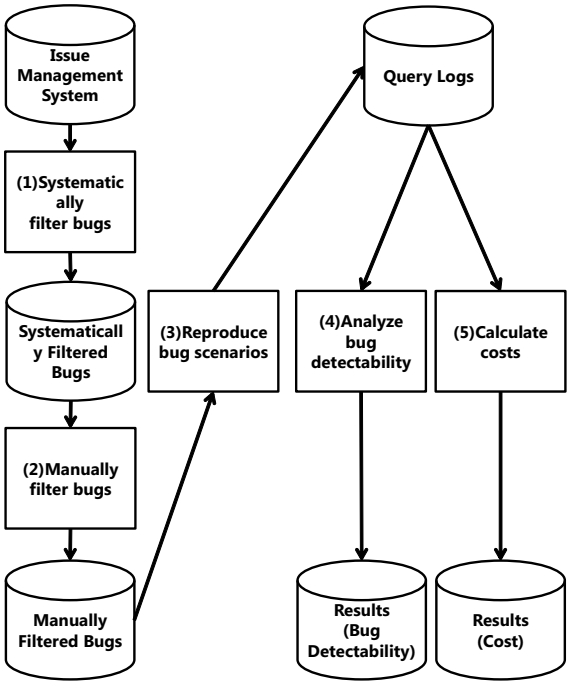

Fig. 5. Overview of the case study

2) Manually filter bugs:
Next issues not related with bugs due to the records in the database and not detected by code coverage are manually filtered. Unrelated issues include problems in setting up software, problems generated by Continuous Integration (CI) tools, etc.

3) Reproduce bugs scenarios:
Each bug is reproduced by conducting its scenarios in order to obtain its query log. Scenarios are determined by inspecting the discussion, activities, and committed files in their issue pages.

4) Analyze bug detectability:
Each bug is then detected by three different methods: Full Predicate Coverage Testing (FPCT) based on full predicate coverage [8] (Section II), PPC Testing (PPCT) based on PPC (Section III), and SPC Testing (SPCT) based on SPC (Section III). To evaluate whether the testing methods can detect the bug, the procedure to execute each method involves three steps. First, which records cause the bug (some columns have certain values, etc.) are determined by inspecting the discussion, activities, and committed files in the issue. Second, records are prepared based on the query log obtained by the reproduction of bug scenarios so that each method achieved 100% coverage. Finally, the applicability of testing method is verified. If a testing method identifies at least one record from the first step, it is deemed as appropriate. The coverage measurement tool depends on the testing method. For FPCT, *SQLFpc*, a full predicate coverage rules generation tool to test SQL database queries [17], is used. Our tools mentioned in Section III are used for PPCT and SPCT. The equivalence

TABLE II
CASE STUDY TARGET SUMMARY

Target	Version	LOC	Tables	Columns	Release Date
OpenMRS	1.8	416446	95	945	Jun., 2011
	1.9	460725	102	1084	Jun., 2012
	1.10	449621	101	1104	Nov., 2014
	1.11	500015	99	1101	Feb., 2015
Broadleaf Commerce	4.0	385369	183	1033	Apr., 2015

TABLE III
TRANSITION OF THE NUMBER OF BUGS

Step	Target				
	OpenMRS				*Broadleaf Commerce*
	1.8	1.9	1.10	1.11	4.0
Initial	453	720	424	379	125
After step 1	61	94	14	29	39
After step 2	20	63	9	11	9

classes of columns are needed from step 4 (Section III). For user definitions, the equivalence classes for each column are set based on the type of column. For a number-type column (e.g., integer, double, float), 'any values in the range for the column' is set as an equivalence class. For a date-type column (e.g., date, datetime), 'past' and 'future', which mean past or future from the present time, respectively, are set. A string-type column (e.g., char, string) is set as either an empty string or a not empty string. Notice that 'NULL' is added as an equivalence class if the column allowes NULL values.

5) Calculate costs:
Finally, the cost of each testing method is calculated. The number of records to reach 100% coverage is regarded as the cost of the testing method. As mentioned in Section III, PPC cannot reach 100% because of redundant parameter combinations. Therefore, we regard the number of records, where all parameter combinations including the redundant ones appear, as the cost of PPCT. The generation tools to reach 100% coverage vary by method. For FPCT, *SQLFpc* is use, whereas *PICT* [14], which is the test generation tool by pairwise testing with the user definitions, is used for PPCT and SPCT.

Table II shows the LOC, number of tables and columns that each version of software interacts with, and the release date of each version. Table III shows the number of bugs at each step.

B. Case Study Results

Table IV shows the results of the bug detectability for each testing method. The first three columns under *Detectability for Each Method* denote the bug patterns. A check (cross) indicates that the test method can (cannot) detect the bug. The fourth through eighth columns denote the number of bugs for each version of *OpenMRS* and *Broadleaf Commerce* by pattern. The last column contains the total number of bugs for each pattern. Table V shows the cost of each testing method. The first and second columns indicate the version of *OpenMRS* and *Broadleaf Commerce*. The third through fifth columns,

TABLE IV
RESULTS OF BUG DETECTABILITY ANALYSIS

Detectability for Each Method			Number of Bugs					Total
			OpenMRS				*Broadleaf Commerce*	
FPCT	PPCT	SPCT	1.8	1.9	1.10	1.11	4.0	
×	×	×	2	15	2	5	3	27
×	×	✓	0	0	0	0	0	0
×	✓	×	0	1	0	0	0	1
×	✓	✓	16	34	3	5	3	61
✓	×	×	0	4	2	0	0	6
✓	×	✓	0	0	0	0	0	0
✓	✓	×	2	3	0	1	0	6
✓	✓	✓	0	6	2	0	3	11
Total			20	63	9	11	9	112

TABLE V
RESULTS OF THE COST CALCULATION

Software	Versions	Number of Records			Cost Ratio (%)	
		FPCT	PPCT	SPCT	SPCT/FPCT	SPCT/PPCT
OpenMRS	1.8	86	288	259	301.16	89.93
	1.9	305	873	711	233.11	81.44
	1.10	88	196	185	210.23	94.39
	1.11	50	149	145	290.00	97.32
Broadleaf Commerce	4.0	90	137	137	152.22	100.00
Total		619	1643	1437	232.15	87.46

Number of Records, contain the number of records needed to reach 100% coverage. The last two columns, *Cost Ratio*, contain the ratio of SPCT to other testing methods.

V. DISCUSSION

A. RQ1: Can SPCT detect bugs that existing testing method cannot?

According to the seventh row in Table IV, SPCT detected 61 bugs that FPCT did not. Furthermore, neither FPCT nor SPCT detected $27+1=28$ bugs (the fourth and sixth rows in Table IV). In total, SPCT detected $61/(61+28)=68.54(\%)$ of the bugs that FPCT could not. Furthermore, according to the sixth row, PPCT detected one more bug than FPCT.

On the other hand, according to the eighth row, FPCT detected six bugs that PPCT and SPCT could not. Furthermore, only ten of the bugs were detected by all three methods. Therefore, our proposed testing methods and FPCT are orthogonal, and should be used in combination to detect more bugs.

The types of bugs detected depend on the detectability patterns. It should be noted that because PPCT involves SPCT, it can detect the same bugs as SCPT. We observed six different patterns with actual issues.

1) Not detected by any of the testing methods:
The bugs in this pattern are not detected by any of the testing methods mainly because the records causing a bug are obtained by multiple queries but each method focuses on one query. This pattern requires that multiple queries be detected simultaneously. Figure 6 shows the actual queries of the issue 'TRUNK-4437', which is an ID of the issue of *OpenMRS*. The table `orders` is related with the table `drug` by the column `drug_inventory_id` as both have the column `concept_id`. The module causes a bug when it is executed with records whose `orders.concept_id` $\neq$

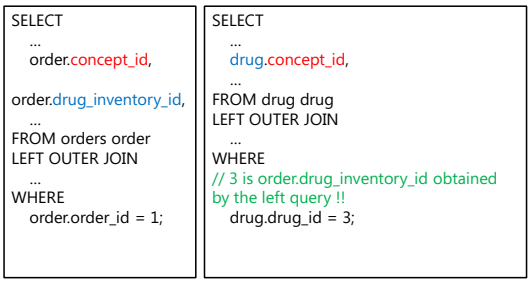

Fig. 6. Actual queries of the issue (TRUNK-4437)

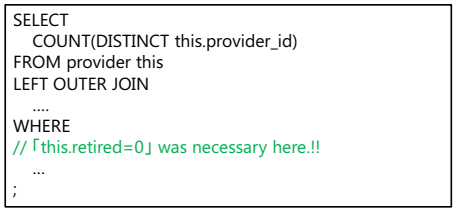

Fig. 7. Actual query of the issue (TRUNK-3339)

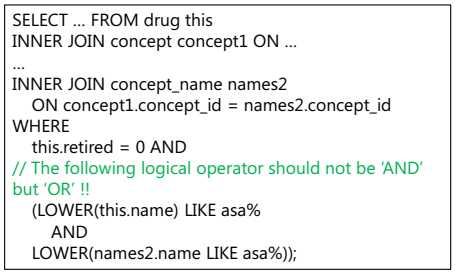

Fig. 8. Actual query of the issue (TRUNK-4530)

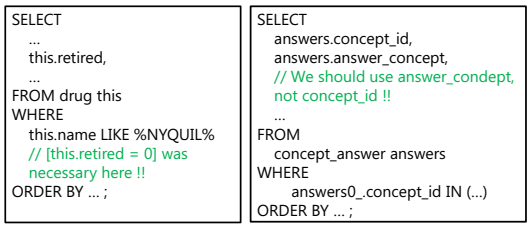

Fig. 9. Actual queries of the issues (TRUNK-4116, TRUNK-3620)

`orders.drug.concept_id`. In Figure 6, a bug is detected when `order.concept_id` obtained by the left query is not the same as `drug.concept_id` obtained by the right query. However, such bugs are not detected because these values are obtained separately. A similar result occurs for the issue '1298' of *Broadleaf Commerce*.

2) Only detected by PPCT:

The bugs in this pattern are detected only by PPCT due mainly to two reasons. First, some of the necessary predicates are omitted in the query. Second, the cause is something other than the value of the record (e.g., the number of records). Figure 7 shows an actual query with the issue 'TRUNK-3339'. Although this query is supposed to return the number of `provider` records obtained by the condition, it returns the number of *not-retired* `provider` records. The module causes a bug when it is executed with *retired* `provider` records. Therefore, an additional condition '`this.retired = 0`' is necessary. FPCT cannot detect such a bug. Furthermore, this query only selects the column `provider_id`. SPCT cannot detect this bug because the query does not select the column `retired`, which causes the bug.

3) Only detected by FPCT:

The bugs in this pattern are detected only by FPCT mainly because there is a misplaced logical operator (AND, OR) or a mistake in the conditional inversion in the query. Figure 8 shows an actual query with the issue 'TRUNK-4530'. This query is supposed to return `drug` records when both the `name` and `name` of the table `concept_name` related with `drug` are forward matches of 'asa'. However, this query actually returns `drug` records when either `name` was a forward match of 'asa'. The module causes a bug when it is executed with `drug` records and either `name` is not a forward match of 'asa'. Therefore, the logical operator 'AND' of the second condition in the WHERE clause should be 'OR'. FPCT can detect such a bug because it can detect misplaced logical operators and conditional inversion mistakes in the query, whereas PPCT and SPCT cannot.

4) Only detected by PPCT and SPCT:

The bugs in this pattern are divided into two types. The first is 'missing predicates', which is due to two reasons; the columns whose values of records cause the bug are selected in the query or some of the necessary predicates are omitted in the query. The left side query in Figure 9 shows an actual query with the issue 'TRUNK-4116'. Although this query is supposed to return `drug` records whose `name LIKE %NYQUIL%`, but it returns *not-retired* `drug` records whose `name LIKE %NYQUIL%`. The module causes a bug when it is executed with *retired* `drug` records whose name `name LIKE %NYQUIL%`. Therefore, the additional condition '`this.retired = 0`' is necessary. FPCT cannot detect such a bug. Both PPCTand SPCT can because the former selects the table `drug`, while the latter selects the column retired of the table drug. We obtained a similar result for the issue '1334' of *Broadleaf Commerce*. The second type is 'misused column values'. PPCT and SPCT detect this type of bug due mainly to two reasons; either the columns whose values of records cause the bug are selected in the query or the value of the wrong column is used in the code. The right side query in Figure 9 shows an actual query with the issue 'TRUNK-3620'. This query selects `concept_id` and `answer_concept` from `concept_answer`. Although the value of `concept_answer` should be used in the code, the value `concept_id` is used instead. The module causes a bug when it is executed with records whose `concept_id` and `answer_concept` have different values (e.g., non-null values for `concept_id` and null values for `answer_concept`).

```
SELECT
   ...
   this.retired,
   ...
FROM drug this
WHERE
// 「this.retired = 0」was redundant !!
   this.retired = 0 AND
   this.name LIKE %2%
ORDER BY ... ;
```

Fig. 10. Actual query of the issue (TRUNK-2392)

```
// At least one record obtained by the query can
cause the bug !!
SELECT
   ...
FROM location_tag_map map0
INNER JOIN location_tag locationta1
   ON map0.location_tag_id =
locationta1.location_tag_id
WHERE
   map0.location_id = 1;
```

Fig. 11. Actual query of the issue (TRUNK-4036)

5) Only detected by FPCT and PPCT:
The bugs in this pattern are detected by FPCT and PPCT mainly because redundant conditions exist in the WHERE clause. Figure 10 shows an actual query with the issue 'TRUNK-2392'. Although this query should return *not-retired* `drug` records whose `name LIKE %2%`, it returns all `drug` records whose `name LIKE %2%`. The module causes a bug when it is executed with *retired* `drug` records whose `name LIKE %2%`. FPCT can detect such a bug because it can notice misplaced logical operators. Similarly, PPCT also can detect such a bug because the table `drug` selected in the query has the column `retired`, which is referred to in the WHERE clause.

6) Detected by any of the three testing methods:
The bugs in this pattern are detected by any of the testing methods mainly because any record in the query causes the bug. Figure 11 shows the actual query of the issue 'TRUNK-4036'. This query should return `location_tag` records related with the `loation` record whose `location_id = 1`. However, the module causes a bug when this query returns at least one `location_tag` record. Because the records are prepared using the query, such a bug can be detected with any of the testing methods. A similar result is obtained for the issue '1310' of *Broadleaf Commerce*.

B. RQ2: How many records are needed to reach 100% SPC coverage?

To reach 100% coverage for *OpenMRS*, 619 records for FPCT, 1643 records for PPCT, and 1437 records for SPCT are needed (Table V, row 8). Thus, SPCT requires about 2.3 more records than FPCT, indicating that SPCT is more costly. However, compared to PPCT, SPCT requires fewer records (0.87 times). For bugs that both PPCT and SPCT detect, SPCT is more efficient. The results indicate that although PPCT and SPCT are valuable because they detect bugs not detected by other methods, their efficiencies could be improved.

C. Limitation

1) Cost Calculation:
We defined the number of records to reach 100% coverage as the cost. However, for PPCT and SPCT, we need make additional efforts to set the candidate equivalence classes of the columns. In our case study, we made very few efforts because we set them uniformly based on the type of columns. In the future, we have to conduct some experiments to measure the additional cost because the equivalence classes of the columns setting takes some time in practical use.

2) Our Coverage Measurement Tool:
We implemented a coverage measurement tool for PPC and SPC, but some parts are incomplete.

To generate the coverage criteria in the step 4 in Section III, our tool analyzes the SQL(SELECT) query to collect the selected tables, columns, and conditions in the WHERE clause, but ignores conditions in the JOIN clause. To reduce the cost, the conditions in the JOIN clause must be collected to remove parameter combinations that do not satisfy the conditions in the JOIN clause. Actually, some queries found in the issues of *Broadleaf Commerce* had many JOIN clauses, but because SPCT only used the simple WHERE clause, it required as many records as PPCT. Furthermore, our tool ignores the condition between columns such as `column1 = column2` in the WHERE clause. When such conditions exists in the query, focusing on the overlapping equivalence classes should reduce the costs.

In the coverage measurement in the step 5 in Section III, our tool inspects which parameter combinations are covered using the query results. However, it regards binary types and enumerated types as string types. In order to calculate PPC and SPC more accurately, both binary and enumerated types should be processed as their own types.

D. Threats to Validity

In the case study, we manually set the equivalence classes of the columns based on the types of columns. However, this may affect the accuracy and costs of our testing methods, and is a threat to internal validity. In the future, especially in regression testing, we would like to set the equivalence classes by analyzing the tendency of the values of columns in a real database.

The targets in the case study were *OpenMRS* and *Broadleaf Commerce*, which have about 400KLOC and interact with about 1000 columns. Although its size is sufficient, *Broadleaf Commerce* has fewer bugs than *OpenMRS*. Therefore, the credibility of the results of *Broadleaf Commerce* is slightly lower than that of *OpenMRS*, and is a threat to internal validity. In the future, we would like to conduct a case study on another version of *Broadleaf Commerce*. A threat to external validity is that our case study is limited to open source software systems in a single software domain. In the future, we would like to evaluate our testing methods by applying them to industrial software with different software domains.

VI. Related Work

Pan et al. proposed a method to generate a database state via dynamic symbolic execution (DSE) [18][19]. Then they leveraged DSE to examine close relationships among host variables, embedded SQL query statements, and branch conditions in the source code. They implemented their approach in Pex, which is a state-of-the-art DSE-based test generation tool. Unlike their study, which focused on increasing code coverage, our work focuses on detecting bugs, that are not detected by testing with the code coverage mentioned in Section II.

Some studies have examined coverage based on an SQL query. Chays et al. proposed a framework to test database applications [20][21][22][23] using database schema integrity constraints, user inputs, and conditions in queries to consider the state of a database. They also implemented a tool to populate a database with meaningful data that satisfy the database constraints. Tuya et al. proposed full predicate coverage to test SQL queries based on modified condition/decision coverage [7][8]. They focused on the conditions in queries to consider the necessary database state; this coverage detected misplaced logical operators or mistakes of conditional inversion. They also proposed a database shrinking technique by using this coverage [9]. Although both works focus on the conditions in the queries, they did not emphasize what is selected in the queries. However, their works provide insight some when considering the variations in the records' values. To detect more bugs in database applications, we would like to combine the above techniques with ours.

Tuya et al. have proposed a tool to generate mutants of SQL database queries to test database applications [24][25]. They proposed four mutation operators for SQL queries: *SC*, *OR*, *NL*, *IR*. *SC* performs mutations on the main clauses, *OR* replaces the operators in the conditions, *NL* handles null values, and *IR* replaces identifiers in the query such as columns and constants. Pan et al. also applied these mutation operators to transform SQL queries into normal program codes, and subsequently generated both effective program inputs and sufficient database states via DSE to kill mutants [26]. Zhou et al. also proposed a mutation testing approach to extend these mutation operators and implemented JDAMA (Java®Database Application Mutation Analyzer) for Java®programs that interact with a database [27]. All of these work focused on SQL queries to test database applications but not the combinations of values for columns selected in SQL queries. Because PPCT and SPCT focus on the variation of column's values selected in SQL queries, they can detect bugs even if some of the necessary conditions are omitted.

Kapfhammer et al. proposed def-use coverage for program interactions with databases [28]. They mapped *create, read, update and delete*, which are the four basic functions of database management systems, into database def-use. They also implemented a coverage-monitoring tool to measure this def-use coverage for various granularities [29]. However, this coverage measures whether defined objects (databases, tables, columns and records) are used but not the variations in the record's values. In contrast, PPCT and SPCT measure the variety in the record's values.

VII. Conclusion

To detect more bugs in database applications, we proposed PPCT and SPCT, which are pairwise coverage-based testing techniques. Our methods focus on the selected elements in SQL queries and apply pairwise testing techniques with equivalence classes of columns from user inputs and conditions in the queries. We also implemented a tool that measures PPC and SPC as well as automates some of the testing process. Then we demonstrated their applicability by applying PPCT and SPCT to two open source software systems, which have more than 400KLOC and interact with about 1000 columns. Although PPCT and SPCT currently have higher costs than existing testing techniques, the case study suggests that their costs can be further reduced.

To measure the costs of candidate equivalence classes of columns setting for PPCT and SPCT, we plan to conduct some subject experiments. Furthermore, we plan to consider database schema integrity constraints and clustering techniques for the values in columns in databases studied by Hashimoto et al. [30] to create initial equivalence classes of columns automatically for the setting cost reduction. Additionally, we would like to apply our techniques to industrial products.

References

[1] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Structural coverage of feasible code," in *Proceedings of the 5th Workshop on Automation of Software Test (AST)*, 2010, pp. 59–66.

[2] K. Sakamoto, K. Shimojo, R. Takasawa, H. Washizaki, and Y. Fukazawa, "Occf: A framework for developing test coverage measurement tools supporting multiple programming languages," in *Proceedings of the 2013 IEEE 6th International Conference on Software Testing, Verification and Validation (ICST)*, 2013, pp. 422–430.

[3] E. Albert, I. Cabanas, A. Flores-Montoya, M. Gómez-Zamalloa, and S. Gutierrez, "jpet: An automatic test-case generator for java," in *Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE)*, 2011, pp. 441–442.

[4] K. K. Aggrawal, Y. Singh, and A. Kaur, "Code coverage based technique for prioritizing test cases for regression testing," *ACM SIGSOFT Software Engineering Notes (SEN)*, vol. 29, pp. 1–4, 2004.

[5] E. Juergens, B. Hummel, F. Deissenboeck, M. Feilkas, C. Schlögel, and A. Wübbeke, "Regression test selection of manual system tests in practice," in *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011, pp. 309–312.

[6] S. Tokumoto, K. Sakamoto, K. Shimojo, T. Uehara, and H. Washizaki, "Semi-automatic incompatibility localization for re-engineered industrial software," in *Proceedings of the 2015 International Conference on Software Testing Verification and Validation (ICST)*, 2014, pp. 91–94.

[7] M. J. Suárez-Cabal and J. Tuya, "Using an sql coverage measurement for testing database applications," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (FSE)*, 2004, pp. 253–262.

[8] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Full predicate coverage for testing sql database queries," *Software Testing, Verification & Reliability (STVR)*, vol. 20, pp. 237–288, 2010.

[9] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Query-aware shrinking test databases," in *Proceedings of the 2nd International Workshop on Testing Database Systems (DBTest)*, 2009, pp. 6:1–6:6.

[10] "Pairwise testing." [Online]. Available: http://www.pairwise.org/

[11] "Openmrs." [Online]. Available: http://openmrs.org/

[12] "Broadleaf commerce." [Online]. Available: http://www.broadleafcommerce.org/

[13] D. R. Kuhn and M. J. Reilly, “An investigation of the applicability of design of experiments to software testing,” in *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW)*, 2002, pp. 91–.
[14] “Pairwise independent combinatorial testing tool.” [Online]. Available: http://blogs.msdn.com/b/nagasatish/archive/2006/11/30/pairwise-testing-pict-tool.aspx
[15] “Git.” [Online]. Available: https://git-scm.com/
[16] “Jira.” [Online]. Available: https://jira.atlassian.com/
[17] “Sqlfpc - generation of full predicate coverage rules for testing sql database queries.” [Online]. Available: http://in2test.lsi.uniovi.es/sqlfpc/?lang=en
[18] K. Pan, X. Wu, and T. Xie, “Generating program inputs for database application testing,” in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2011, pp. 73–82.
[19] K. Pan, X. Wu, and T. Xie, “Guided test generation for database applications via synthesized database interactions,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, pp. 12:1–12:27, 2014.
[20] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker, “A framework for testing database applications,” in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2000, pp. 147–157.
[21] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, “An agenda for testing relational database applications: Research articles,” *Software Testing, Verification & Reliability (STVR)*, vol. 14, pp. 17–44, 2004.
[22] Y. Deng, P. Frankl, and D. Chays, “Testing database transactions with agenda,” in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 78–87.
[23] D. Chays, J. Shahid, and P. G. Frankl, “Query-based test generation for database applications,” in *Proceedings of the 1st International Workshop on Testing Database Systems (DBTest)*, 2008, pp. 6:1–6:6.
[24] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva, “Sqlmutation: A tool to generate mutants of sql database queries,” in *Proceedings of the 2nd Workshop on Mutation Analysis (Mutation)*, 2006, pp. 1–.
[25] J. Tuya, M. J. Suárez-Cabal, and C. d. la Riva, “Mutating database queries,” *Information and Software Technology*, vol. 49, pp. 398–417, 2007.
[26] K. Pan, X. Wu, and T. Xie, “Automatic test generation for mutation testing on database applications,” in *Proceedings of the 8th International Workshop on Automation of Software Test (AST)*, 2013, pp. 111–117.
[27] C. Zhou and P. Frankl, “Mutation testing for java database applications,” in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation (ICST)*, 2009, pp. 396–405.
[28] G. M. Kapfhammer and M. L. Soffa, “A family of test adequacy criteria for database-driven applications,” in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2003, pp. 98–107.
[29] G. M. Kapfhammer and M. L. Soffa, “Database-aware test coverage monitoring,” in *Proceedings of the 1st India Software Engineering Conference (ISEC)*, 2008, pp. 77–86.
[30] Y. Hashimoto, K. Oshima, H. Danno, R. Mibe, and K. Yamaguchi, “A proposal of a method to analyze rdb by columns to understand details of the specification,” *The National Convention of IEEJ Electronics, Information and Systems*, pp. 171–175, 2013.